(19)日本国特許庁(JP)　(12)特　許　公　報(B2)　(11)特許番号
特許第4249267号
(P4249267)
(45)発行日　平成21年4月2日(2009.4.2)　(24)登録日　平成21年1月23日(2009.1.23)

(51)Int.Cl.　F I
G06F 12/00 (2006.01)　G06F 12/00 501H
G06F 12/00 501J

請求項の数 25　(全 22 頁)

(21)出願番号　特願平10-538908
(86)(22)出願日　平成10年3月6日(1998.3.6)
(65)公表番号　特表2001-506387(P2001-506387A)
(43)公表日　平成13年5月15日(2001.5.15)
(86)国際出願番号　PCT/US1998/004567
(87)国際公開番号　WO1998/039769
(87)国際公開日　平成10年9月11日(1998.9.11)
審査請求日　平成17年1月13日(2005.1.13)
(31)優先権主張番号　08/813,621
(32)優先日　平成9年3月7日(1997.3.7)
(33)優先権主張国　米国(US)

(73)特許権者
マイクロソフト　コーポレイション
アメリカ合衆国　９８０５２　ワシントン州　レッドモンド　ワン　マイクロソフト　ウェイ（番地なし）
(74)代理人
弁理士　谷　義一
(74)代理人
弁理士　阿部　和夫
(72)発明者　ショーロフ，スリカンス．
アメリカ合衆国　９８０２９　ワシントン州　アイザッカー　サウスイースト　４２　エヌディー　ストリート　２５０４９

最終頁に続く

(54)【発明の名称】ファイル・システムにおけるディスク・スペースの解放

(57)【特許請求の範囲】
【請求項１】
持続的記憶媒体内のファイルに割り振られた記憶スペースを解放する方法であって、ファイル・システムは、前記ファイルを前記記憶媒体内の物理記憶スペース・ロケーションと関連づけるマッピング・データを維持する方法において、
前記ファイルと、ファイルの論理的先頭または論理的末尾を含んでいない、前記ファイル内の選択された論理データ・ブロックとを特定する情報を前記ファイル・システムに与えるステップと、
前記情報を前記ファイル・システムで受け取り、前記マッピング・データにアクセスして、前記選択された論理データ・ブロックに対応する、前記持続的記憶媒体内の少なくとも１つの物理ロケーションを決定するステップと、
前記少なくとも１つの物理ロケーションがもはや前記ファイルに割り振られていないことを示すように前記ファイル・システム内の前記マッピング・データを修正するステップとを備えることを特徴とする方法。
【請求項２】
請求項１に記載の方法において、前記少なくとも１つの物理ロケーションは、そのアイデンティティに対応する第１インジケータを前記マッピング・データ内にもち、前記ファイル・システム内の前記マッピング・データを修正する前記ステップは、前記マッピング・データ内において前記第１インジケータに代えて第２インジケータを書き込み、前記ファイルと関連づけて前記第２インジケータを維持するステップを備えることを特徴とする方

法。
【請求項３】
請求項１に記載の方法において、前記ファイル・システムは、どのファイルにも割り振られていない空きスペースである物理ロケーションを表す空きスペース情報を維持し、前記方法は、前記少なくとも１つの物理ロケーションが空きスペースであることを示すように前記空きスペース情報を修正するステップをさらに備えることを特徴とする方法。
【請求項４】
請求項３に記載の方法において、前記空きスペース情報は、ディスク上の前記物理ロケーションに対応するビットをもつビットマップに維持され、前記空きスペース情報を修正する前記ステップは、前記少なくとも１つの物理ロケーションに対応する少なくとも１つのビットの値を変更するステップを備えることを特徴とする方法。
【請求項５】
請求項１に記載の方法において、前記選択された論理データ・ブロックの固定サイズを決定するステップをさらに備えることを特徴とする方法。
【請求項６】
請求項１に記載の方法において、情報を前記ファイル・システムに与える前記ステップは、アプリケーション・プログラミング・インタフェースをコールするステップを含むことを特徴とする方法。
【請求項７】
請求項１に記載の方法において、前記ファイルを特定する情報を前記ファイル・システムに与える前記ステップは、ファイル・ハンドルを前記ファイル・システムに与えるステップを含むことを特徴とする方法。
【請求項８】
請求項１に記載の方法において、論理データ・ブロックを特定する情報を前記ファイル・システムに与える前記ステップは、前記ブロックのサイズを表すサイズ値を与えるステップを含むことを特徴とする方法。
【請求項９】
請求項８に記載の方法において、前記サイズ値はバイト数であり、前記情報を前記ファイル・システムで受け取り、前記マッピング・データにアクセスする前記ステップは、前記バイト数をアロケーション単位数に変換するステップを含むことを特徴とする方法。
【請求項１０】
請求項１に記載の方法において、論理データ・ブロックを特定する情報を前記ファイル・システムに与える前記ステップは、前記ファイル内の前記ブロックの論理ロケーションを表すオフセット値を与えるステップを含むことを特徴とする方法。
【請求項１１】
請求項１０に記載の方法において、前記オフセット値はバイト数であり、前記情報を前記ファイル・システムで受け取り、前記マッピング・データにアクセスする前記ステップは、前記論理ロケーションを特定のアロケーション単位に変換するステップを含むことを特徴とする方法。
【請求項１２】
請求項１に記載の方法において、前記ファイルの前記マッピング・データは、レコードのマスタ・ファイル・テーブル内の前記ファイルに対応するレコード内に維持されることを特徴とする方法。
【請求項１３】
請求項１に記載の方法において、アロケーション単位はクラスタであり、前記ファイルの前記マッピング・データはクラスタ・ランのエクステント・リスト内に維持されることを特徴とする方法。
【請求項１４】
請求項１に記載の方法において、アロケーション単位はクラスタであり、前記マッピング・データはクラスタのブロックリスト内に維持されることを特徴とする方法。

【請求項１５】
データをストアするための複数の物理ロケーションをもつ永続的記憶媒体を含むコンピュータ・システムであって、前記コンピュータ・システムは、論理データのファイルに割り振られた物理記憶スペースを解放するように構成され、
マッピング・データをもつファイル・システムであって、該マッピング・データは前記ファイルの前記論理データを前記記憶媒体の物理ロケーションと関連づけるファイル・システムと、
ファイルの論理的先頭または論理的末尾を含んでいない、前記ファイルに割り振られた論理データのセクションを解放する要求を受け取る手段と、
前記要求を処理する手段であって、前記マッピング・データにアクセスして前記論理データのセクションに対応する少なくとも１つの物理ロケーションを決定する手段と、該少なくとも１つの物理ロケーションを前記ファイルの前記論理データから切り離すように前記マッピング・データを修正する手段とを含む手段と
を備えることを特徴とするコンピュータ・システム。
【請求項１６】
請求項１５に記載のコンピュータ・システムにおいて、前記少なくとも１つの物理ロケーションはそのアイデンティティに対応する第１インジケータを前記マッピング・データの中にもち、前記マッピング・データを修正する前記手段は第１インジケータを第２インジケータでオーバライトする手段を含むことを特徴とするコンピュータ・システム。
【請求項１７】
請求項１５に記載のコンピュータ・システムにおいて、前記ファイル・システムはどのファイルにも割り振られていない空きスペースである物理ロケーションを表す空きスペース情報を維持し、前記要求を処理する前記手段は前記少なくとも１つの物理ロケーションが空きスペースであることを示すように前記空きスペース情報を修正する手段を含むことを特徴とするコンピュータ・システム。
【請求項１８】
請求項１７に記載のコンピュータ・システムにおいて、前記記憶媒体はディスクであり、前記空きスペース情報は、前記ディスクに関連してストアされるビットマップに維持されることを特徴とするコンピュータ・システム。
【請求項１９】
請求項１５に記載のコンピュータ・システムにおいて、前記記憶媒体の前記物理ロケーションはディスク上にクラスタを備え、前記論理データのセクションは少なくとも１つのクラスタに対応することを特徴とするコンピュータ・システム。
【請求項２０】
請求項１５に記載のコンピュータ・システムにおいて、前記論理データのセクションは、バイトで表したオフセット値とそのあとに続くバイト数とで特定されることを特徴とするコンピュータ・システム。
【請求項２１】
請求項１５に記載のコンピュータ・システムにおいて、前記ファイルの前記マッピング・データはレコードのマスタ・ファイル・テーブル内の前記ファイルに対応するレコード内に維持されることを特徴とするコンピュータ・システム。
【請求項２２】
請求項１５に記載のコンピュータ・システムにおいて、アロケーション単位はクラスタであり、前記ファイルの前記マッピング・データはクラスタ・ランのエクステント・リスト内に維持されることを特徴とするコンピュータ・システム。
【請求項２３】
請求項１５に記載のコンピュータ・システムにおいて、アロケーション単位はクラスタであり、前記マッピング・データはクラスタのブロック・リスト内に維持されることを特徴とするコンピュータ・システム。
【請求項２４】

請求項１５に記載のコンピュータ・システムにおいて、前記要求を処理する前記手段はアプリケーション・プログラミング・インタフェースを含むことを特徴とするコンピュータ・システム。
【請求項２５】
請求項２４に記載のコンピュータ・システムにおいて、前記アプリケーション・プログラミング・インタフェースは、複数のソース・ファイルを少なくとも１つのターゲット・ファイルに順次にマージするアプリケーション・プログラムによってコールされることを特徴とするコンピュータ・システム。
【発明の詳細な説明】
発明の分野
本発明は一般的にはファイル・システムに関し、さらに具体的には、ファイル・システムによって割り振られたディスク・スペースを解放する改良方法およびメカニズムに関する。
発明の背景
ファイル・システムが受け持つ仕事の１つは、ファイル内の論理データと、そこにデータがストアされている永続的記憶ボリューム（permanent storage volume）上に置かれている物理アロケーション（割振り）単位（例えば、クラスタ）との間の関係をマッピング（対応づけること）することである。ファイル内の有用データ量はそのサイズが縮小されると、そのファイルを取り扱うアプリケーション・プログラムはファイル・サイズが縮小したことをファイル・システムに通知し、ファイルに割り振られたディスク・スペースの一部を解放して再使用できるようにする。解放するデータがファイルの前方にあるときは、残余データをファイルの先頭に移し、ファイルの前方を基準にした新しいファイル・サイズをファイル・システムに知らせるのは、アプリケーション・プログラムの責任である。ファイル・システムはファイルの末尾にマッピングされたクラスタを空きスペースに戻すことで、基本的には不必要な内容をファイルの末尾から削除することによってスペースを解放する。
しかし、多くのアプリケーションは順次に、つまり、前方から後方へ向かう順序でデータを処理していく。例えば、マージ・アプリケーションでは、２つまたはそれ以上のソート・ファイルは単一のソート・ターゲット・ファイルにマージされ、その時点でソース・ファイルは必要でなくなる。このようなマージは、ソース・ファイルの各々からデータを順次に処理していき、そのデータを該当のソート順序に従って結合し、その結合データをソート・ターゲット・ファイルに書き込むことによって行われている。マージ・アプリケーション・プログラムは大きなファイル（例えば、５００メガバイト）をマージすることがよくあるので、ソース・ファイル読取りとターゲット・ファイル書込みは、ソース・データのすべてが処理されるまでデータについて少量ずつ繰り返し行われている。ソース・データの処理が完了すると、ソース・ファイルは削除されるのが一般である。
上述したマージング方法は非常に簡易化されているが、その操作期間に大量の空きディスク・スペースが残っていることが必要である。例えば、ソース・ファイルの結合サイズの総計が５００メガバイトであれば、ターゲット・ファイルもサイズが５００メガバイトになる可能性がある。マージを行うためには、最大５００メガバイトのディスク・スペースの空きがなければ、ソース・ファイルを削除することができない。このことは、ソース・ファイルがいったん削除されたあとも、総占有ディスク・スペースは未変更になっているのが一般であるので、空きディスク・スペースが基本的に一時的なものであっても同じである。もちろん、ターゲット・ファイルは一部の重複データが除去されていれば、ソース・ファイルより小さくなる場合がある。以上から理解されるように、このような大量の空きスペースはどのディスク・ボリューム上でも常に残っているとは限らない。さらに、アプリケーション・プログラムに大量のデータを定期的に各ファイルの前方に移させて、そのデータが消費されるときソース・ファイルを後方から縮小できるようにすることは非常に非効率である。
上述した一時スペースの問題を解決するために、ファイルをマージする別の方法は、一緒

にすると大きな論理ファイルを構成する、複数の小さなファイルを管理するマージ・アプリケーション・プログラムを書くことである。このアプリケーション・プログラムは、小さなファイルが大きなファイルを構成していく様子を追跡し、データの処理時に空きディスク・スペースを作るように一部のファイルを削除することを管理する。しかし、大きな論理ファイルを構成する複数のファイルを管理することは非常に複雑である。例えば、プログラムは大きなソース・ファイルを小さなファイルに分割し、各ファイルに名前を付け、各ファイル間の論理関係を維持する必要があるため、基本的にはファイル・システム内のファイル・システムの働きをすることになる。さらに、大部分のオペレーティング・システムはアプリケーションがもつことができる、同時オープン・ファイルの数を制限しているため、同時オープン・ファイルの数が多くなるとパフォーマンスが犠牲になる。オープン・ファイルが余りに多くなるのを防止するには、アプリケーション・プログラムがさらに複雑化することになる。

前方から後方への順序でデータを同じように処理する他のアプリケーションとしては、先入れ先出し（ＦＩＦＯ）キューのファイルを取り扱うアプリケーションがある。この種のキューでは、新しいアイテムはキューの最後に追加され、必要でなくなったアイテムはキューの前方から除去される。従って、ＦＩＦＯキューは、新しいアイテムをキューの最後に追加するEnQueueオペレーションと、キューが空でなければキューの前方からアイテムを除去するDeQueueオペレーションをサポートしている。IsEmptyオペレーションも用意されているが、これはキューが空であるかどうかをテストするものである。

個々のアイテムをデキュー（Dequeuing：キューから除くこと）することは持続的ＦＩＦＯキュー（persistent FIFO queue）、つまり、ディスクなどの永続的記憶媒体上にストアされたＦＩＦＯキューでは高価になる。この費用が発生するのは、デキューされたアイテム（以下、「デキュー・アイテム」という）をファイルから消去するために非常に多数の高価な入出力ディスク・オペレーションが必要になるためである。事実、持続的ＦＩＦＯキューでは、各アイテムがデキューされた直後に各アイテムをファイルから消去するのではなく、キューをクリーンアップするプログラムは複数のデキュー・アイテムをまず累積し（その際、これらのアイテムを記憶している）、そのあとでこれらのアイテムをバルクでファイルから消去している。デキュー・アイテムをこのようにバッチで消去するようにすると、複数のデキュー・オペレーションにわたるデキュー・コストが償却されることになる。

デキューされた（しかし、消去されていない）アイテムと、デキューされなかった残余アイテム（以下「非デキュー・アイテム」という）の両方を収めている持続的ＦＩＦＯキュー・ファイルをクリーンアップする方法はいくつかが知られている。最初の方法では、デキュー・データに残余データをオーバライト（重ね書き）している。つまり、残余データをファイルの前方に移し、そのあと残余データのサイズに基づいてファイル・サイズを縮減している。これは、残余データのサイズに等しい一時ファイルを作成し、残余データをその一時ファイルにコピーし、そのあと残余データのコピーをファイルの前方からとってオリジナル・ファイルに戻すことによって行われている。一時ファイルはそのあとで削除されている。

第２の方法は第１の方法と似ているが、一時ファイルのコピーをオリジナル・ファイルに戻すのではなく、一時ファイルが新しい持続的ＦＩＦＯキュー・ファイルとなり、古いＦＩＦＯキュー・ファイルは削除されている。ファイル・システムは必要に応じて、ファイル・ヘッダ情報を新しいＦＩＦＯキュー・ファイルの名前で改名し、あるいは更新する。しかし、第１と第２の方法のどちらも、一時ディスク・スペースはデキューされなかったデータのサイズと等しいものを作る必要がある。さらに、第１と第２の方法では、潜在的に大量のデータがコピーされることになり、データをコピーすることは非常に費用がかかっている。

第３の方法では、非デキュー・データをファイル自体の中でファイルの前方に移すことによってデキュー・データに非デキュー・データをオーバライトしている。しかし、この方法によると一時空きスペースが不要になるが、データを移動するために大量のデータをコ

ピーすることが依然として行われている。さらに、コピーをとっている途中でシステムに障害が起こると、ファイルが不整合の状態になるおそれがある。
最後に、アイテムは最初のファイルから最後のファイルまでにわたってシリアル番号を付けた、複数の小さなファイル内に保存しておくことができる。新しいアイテムは最後のファイルに付加されていき、そのファイルが一杯になると、新しいファイルが作成され、そのファイルが最後のファイルとなるため、ファイル総数が増加していく。最初のファイル内のアイテムがすべてデキューされると、最初のファイルは削除され、そのファイルのスペースがファイル・システムに戻される。以上から理解されるように、この方法では、複雑に階層化された、余分のファイル管理ソフトウェアを開発し、保守する必要がある。

発明の目的と概要

従って、本発明の目的は、一般的には、ファイル・システムにおいてあるファイルの任意の論理部分に割り振られたディスク・スペースを解放する方法およびメカニズムを提供することである。
さらに具体的には、本発明の目的は、ディスク・スペースの解放を高速化すると共に、大量のデータをコピーしたり、大量の一時スペースを割り振ったりしないで済むようにする、かかる方法およびメカニズムを提供することである。
本発明の別の目的は、ファイル・システムに組み込まれている、上記種類に属する前記方法およびメカニズムを提供することである。
本発明の関連目的は、システム障害から保護するためにファイル・システムが具備する既存のセーフガードと共に機能する方法およびメカニズムを提供することである。
上記目的を達成するために、本発明の関連目的はマージ・アプリケーションや持続的ＦＩＦＯキューを取り扱うアプリケーションなどの、アプリケーション・プログラムで容易に利用することができる、上記特徴を具備した方法およびメカニズムを提供することである。
本発明のさらに別の目的は、拡張可能で、ほとんどどのファイル・システムでも有効に働く、高速で、単純な、信頼性のある方法およびメカニズムを提供することである。
以上を要約すると、本発明は持続的記憶媒体内のファイルに割り振られた記憶スペースを解放する方法およびメカニズムを提供している。ファイル・システムはファイルを記憶媒体内の物理記憶スペース・ロケーションと関連づけているマッピング・データを維持している。アプリケーション・プログラムなどは、ファイルおよびそのファイル内で選択された論理データ・ブロックを特定する情報をファイル・システムに与える。ファイル・システムはこの情報を受け取ると、マッピング・データにアクセスし、選択された論理データ・ブロックに対応する、永続的記憶媒体内の物理ロケーションを決定する。ファイル・システムは、その物理ロケーションがもはやそのファイルに割り振られていないことを示すようにマッピング・データを修正し、その物理ロケーションを空きスペースに追加する。
本発明の方法およびメカニズムをマージ・アプリケーションで使用すると、ソース・ファイルのデータが大きなターゲット・ファイルに追加されたときソース・ファイルを縮小することも、あるいは持続的ＦＩＦＯキューからスペースをデキューすることもできる。
本発明のその他の目的と利点は添付図面を参照して、以下の詳細な説明の中で明らかにする。

【図面の簡単な説明】

図１は、本発明を組み込むことができるコンピュータ・システムを示すブロック図である。
図２は、ファイル・システムのメタデータをディスク・ボリューム上にストアするためのテーブルを示す図である。
図３と図４はそれぞれ、本発明の一側面に従ってディスク・スペースが解放される前と後のエクステント・リストを示す図である。
図５は、本発明の一側面に従って修正される途中にある図３のエクステント・リストを示す図である。
図６は、修正された後の図５のエクステント・リストを示す図である。

図７は、本発明の一側面に従って修正された後の図２のテーブルを示す図である。
図８は、修正された後の図３のエクステント・リストを示す図である。
図９は、最適化された後の図８のエクステント・リストを示す図である。
図１０と図１１はそれぞれ、代替ファイル・システムにおいてディスク・スペースが解放される前と後のブロック・リストを示す図である。
図１２は、本発明の一側面に従ってディスク・スペースを解放するときにとられる一般的ステップを示すフロー図である。
図１３乃至図１６は、ソース・ファイルがターゲット・ファイルにマージされるときのシーケンスを示す図である。
図１７は、図１３乃至図１６に示すようにファイルをマージするときにとられる一般的ステップを示すフロー図である。
図１８乃至図２０および図２１乃至図２３は、デキューされ、解放されるデータと共に処理されるＦＩＦＯキューを示す図である。
図２４は、図１８乃至図２０および図２１乃至図２３に示すようにデキュー・スペースを解放するときにとられる一般的ステップを示すフロー図である。

好適実施例の詳しい説明

以下では、添付図面を参照して説明する。まず、図１を参照して説明すると、図１は本発明を組み込むことができるコンピュータ・システム全体を符号２０で示している。図示のコンピュータ・システム２０はサーバでも、ワークステーションでも、またはその組み合わせでもよく、公知のようにコンピュータをベースとする、１つまたは２つ以上の他のリソース（資源）に接続することができる。もちろん、以下の説明で明らかにされるように、本発明はどの特定タイプのコンピュータまたはネットワーク・アーキテクチャにも限定されず、スタンドアロン型パーソナル・コンピュータなどに組み込むことが可能である。
図１に示すように、コンピュータ・システム２０は、オペレーティング・システム２６がそこにロードされているメモリ２４に接続されたプロセッサ２２を搭載している。オペレーティング・システム２６はMicrosoft CorporationのWindows NT™オペレーティング・システムであることが好ましい。コンピュータ２０は、Windows NT™ファイル・システム（ＮＴＦＳ）などのファイル・システム２８をもち、これはオペレーティング・システム２６と関連づけられているか、その内部に組み込まれている。メモリ２４内のアプリケーション・プログラム３０はアプリケーション・プログラミング・インタフェース（Application Programming Interface-API）を通してオペレーティング・システム２６およびファイル・システム２８とのインタフェースとなっている。
また、コンピュータ・システム２０は、コンピュータ・システムを１つまたは２つ以上のネットワーク内デバイスに、キーボードおよび／またはマウスなどの１つまたは２つ以上の入力デバイス３６に、および／またはモニタおよび／またはスピーカなどの１つまたは２つ以上の出力デバイス３８に接続するための入出力（Ｉ／Ｏ）回路３４も実装している。さらに、コンピュータ・システム２０はハードディスク・ドライブ４０などの永続的記憶媒体も実装している。
ファイル・システム２８はハードディスク４０上のファイルを管理し、（１）ファイルをストア、参照、共有および安全保護し、（２）ファイル・データにアクセスし、（３）ファイル保全性を維持するための、メソッドを収容しているのが一般的である。しかしながら、ファイル・システムとその関連のオペレーティング・システムは、特にこれらのファイル・システムがオペレーティング・システムに組み込まれている場合には、常に明確に区別されているとは限らない。従って、以下でファイル・システム２８に属するとしているプロセスまたはステップはいずれも、あるいはすべてはオペレーティング・システム２６でも実行することが可能であるものとし、その逆も同じである。
周知のように、ディスク・ドライブ４０は複数のファイル42_1－42_nをストアしており、これらのファイルには、論理編成のデータがディスク・ドライブ４０上の各所ロケーションに物理的に分散されたアロケーション単位にストアされている。アロケーション単位とは任意のディスク・ボリューム上の基本的記憶単位であり、１つまたは２つ以上のディ

スク・セクタからなるクラスタである場合がある。分散ファイル部分を論理的に連続するデータ・ブロックとして維持するために、ＭＳ－ＤＯＳのＦＡＴ（File Allocation Table-ファイル・アロケーション・テーブル）システム、Microsoftのオブジェクト・ファイル・システム（Object File System-OFS）およびＮＴＦＳなどのファイル・システム２８は各ファイルのメタデータ（各ファイルに関連するアロケーション単位を順序づけている）と関連づけられたマップをストアしている。

図２は、ファイル４２$_1$－４２$_n$がＮＴＦＳファイル・システムでどのような編成になっているかを示す概要図であるが、これについては、文献「Windows NTファイル・システムの内部（Inside the Windows NT File System）」H. Custer著、Microsoft Press, 1994に記載されている。ＮＴＦＳでは、アロケーション単位はクラスタであり、任意のＮＴＦＳボリュームに関する情報はレコードの集合として編成されたマスタ・ファイル・テーブル５０に収められている。例えば、マスタ・ファイル・テーブルはボリュームの各クラスタごとに１ビットを持つビットマップ・レコード５２を収めており、各ビットはクラスタがファイルに割り振られているか、空きスペースであるかを示す値を持っている。さらに、ＮＴＦＳボリューム上の各ファイル４２$_1$－４２$_n$に関する情報はレコード５４$_1$－５４$_n$に収められている。他方、各レコード５４$_1$－５４$_n$はデータ・フィールド５６$_1$－５６$_n$を含む複数のフィールドから構成され、そこにはクラスタ・マッピング情報が入っている。しかし、ＮＴＦＳでは、ファイルを構成する個々のクラスタの順序づけリストをメタデータにストアさせるのではなく、各ファイル４２$_1$－４２$_n$のクラスタの連続するラン（run）を追跡している。このようにすると、メタデータのスペースが節減されるが、これは、大きなファイル内のクラスタの数がそれ自体大きな数であり、ＮＴＦＳはファイルのクラスタを可能な限り連続させることを試みるのが一般であるからである。

図２に示すように、４２$_1$のような、ファイル内の連続クラスタの各ランごとに、データ・フィールド５６$_1$はファイルの仮想クラスタ番号を表している第１の数、ディスク上のクラスタの物理ロケーションを表している第２の数、およびそのランに連続クラスタがいくつあるかを表している第３の数を含んでいる。例えば、図２に示すように、レコード５４$_1$内のデータ・フィールド５６$_1$は、実クラスタ１０にマッピング（対応づけること）された仮想クラスタ０から始まって、１１個の仮想クラスタをファイル５６$_1$にマッピングしている。データは５個の連続するデータ・クラスタにわたっており、仮想クラスタ０－４は実クラスタ１０－１４に対応づけられている。仮想クラスタ５はファイル・データを収めている次のクラスタであり、第２のランに示すように実クラスタ１９にマッピングされている。この第２のランは２の長さにわたっているので、仮想クラスタ６はこのランによって実効的に実クラスタ２０にマッピングされている。データ・フィールド５６$_1$内の他のエントリをたどっていくと、レコード５４$_1$に保存されている１１個の物理クラスタがファイル４２$_1$内の論理編成データにどのようにマッピングされているかが理解される。

クラスタ・ランはファイルのエクステント・リストと呼ばれるものを実効的に構成しており、このリストは図３に示すように、少なくとも各ラン内の最初の実クラスタ番号と、任意のファイル内の各不連続クラスタごとのクラスタ・ランの長さとを収めている。以上から理解されるように、かかるエクステント・リスト５８のすべてまたは一部は各オープン・ファイル（ファイル４２$_1$など）ごとにメモリ２４内のファイル・バッファに置いておくと、ファイル・システム２８はクラスタ・マッピング情報に高速アクセスすることが可能になる。これとは別に、ファイル・システム２８がマスタ・ファイル・テーブル５０のデータ・フィールド５２内のメタデータを直接に利用して、必要とするファイル情報を得ようとすると、エクステント・リスト５８は余分なものになる。

しかし、以下では、説明を簡単にするために、エクステント・リストを中心にして本発明を説明することにする。例えば、ファイル４２$_1$に、従ってデータ・フィールド５６$_1$に対応するエクステント・リスト５８（図３）は同じように、ファイル４２$_1$が４つのランの実クラスタ・ロケーションから構成され、これらは５の長さにわたって（つまり、１０から１４までにわたって）クラスタ１０から始まり、２の長さにわたってクラスタ１９に続

き、１の長さにわたってクラスタ２００に続き、クラスタ５０から３つのクラスタ（クラスタ５０から５２まで）のランで終わっていることを示している。
本発明の一側面によれば、以下で詳しく説明するように、アプリケーション・プログラム３０はファイルのデータの任意のセクションを解放、つまり、デコミット（decommit）することができ、ファイルの末尾だけでスペースを解放することに限定されない。スペースの解放を行うには、アプリケーション・プログラム３０はファイル・システム２８にコールを出し（ＡＰＩ　３２内の定義されたＡＰＩ　４６から）、必要でなくなったが、まだファイルに割り振られたままのデータのセクションをデコミットする。好ましくは、データのセクションは１つまたは２つ以上のクラスタに対応する事前定義サイズの倍数になっている。
スペースを解放するには、ファイル・システム２８はアプリケーション・プログラムからほとんど見えないような形でファイルのメタデータを操作する。具体的には、アプリケーション・プログラム３０は、デコミットされたスペースがファイルの一部としてまだ残っているものとして、その内部ポインタとオフセットを維持している。もちろん、デコミットされたデータ・ブロックがファイルに属さなくなると、そのファイルに読み書きできなくなるが、アプリケーション・プログラム３０はスペースのデコミット前にそのデータまたはスペースが必要でなくなったことを確かめること以外は、なにも調整を行う必要はない。以下で詳しく説明するように、アプリケーション・プログラムがデコミットされたスペースの読み取りを試みると、ゼロが戻されることがあり、書き込みを試みると、デコミットされたスペースが再度デコミットされることがある。
次に、本発明の動作について説明すると、本発明のＡＰＩ　４６がまずアプリケーション・プログラム３０からコールされるとき、プログラム３０はデコミットすべきスペースを含んでいるファイルのハンドル（例えば、ハンドルが５であるファイル42_1）を含む情報をＡＰＩ　４６に渡す。アプリケーション・プログラム３０からは、デコミットを開始する論理ロケーションまでのオフセット（例えば、ファイルの先頭からのバイト数を示すオフセット・ポインタ）とデコミットする長さの値（つまり、バイト数などのスペース量）も渡される。例えば、ファイル42_1のコールでは、５に等しいファイル・ハンドル、１メガバイトのオフセットおよび５１２キロバイトの長さを渡すことができる。好適実施例では、ファイル・ハンドルはオープン・ファイルに対応し、それ以外はＡＰＩ　４６からエラー・メッセージが戻される。
単純性を保つために、バイト数で表したオフセットと長さの値は、好ましくはファイル・セクションと呼ばれる単位の倍数になっている。一般的に、ファイル・セクションのサイズは任意であるが、ファイル・システム２８内では固定されており、ファイル・システム２８がどのようにデータを編成するかと若干の関係をもって選択されている。例えば、ＮＴＦＳでは、ファイル・セクションのサイズはディスク・ボリューム上のクラスタのサイズの倍数であり、試みとして６４キロバイトが選択されている。この６４キロバイトのサイズは５１２バイト・クラスタ、１キロバイト・クラスタ、２キロバイト・クラスタ、以下同様の倍数であり、これらはいずれもＮＴＦＳで使用可能である。ＯＦＳでは、ファイル・セクション・サイズは２５６キロバイトである。このサイズも、ディスク４０の入出力オペレーション数が、各オペレーションで解放されるスペース量とバランスがとられるように選択されている。つまり、多数の小さなデコミットを行わないで済むだけ十分に大きいが、デコミットを行う前に大量の一時ディスク・スペースが必要でないくらい十分に小さくなっている。もちろん、以下で明らかにするように、アプリケーション・プログラム３０は１回のＡＰＩコールで２つ以上のファイル・セクションをデコミットすることが可能である。例えば、１回のＡＰＩコールで、８個の６４キロバイト・ファイル・セクションに相当する５１２キロバイトをＡＰＩ　４６にデコミットさせることが可能である。
アプリケーション・プログラム３０がＡＰＩ　４６をコールしたとき、オフセットおよび／または長さがファイル・セクションの倍数になっていないと、ＡＰＩ　４６からエラー・メッセージなどが戻される。しかし、適切なアプリケーション・プログラム３０は、ファイル・セクションのサイズが事前に知らされているか、あるいはファイル・システム２

８に問い合わせてファイル・セクションのサイズを知ることができるようになっている。従って、別の方法として、オフセットおよび長さの値の単位としてバイト数ではなく、ファイル・セクションをアプリケーション・プログラム３０が渡すようにすることも可能である。

上記とは別に、より複雑ではあるが、ファイル・セクションのサイズに対応する値ではなく、そのオフセットと長さパラメータの任意のバイト値をアプリケーション・プログラム３０が渡すようにすることも可能である。このような代替方法では、ファイル・システム２８はオフセットと長さをファイル・セクションに変換し、可能な限り多数のファイル・セクション（ゼロであることも可能）をデコミットする。次に、ファイル・システム２８はデコミットされなかったバイト数に相当する残余をアプリケーション・プログラム３０に戻すか、さもなければデコミットのエクステントを示すポインタを戻す。そのあと、アプリケーション・プログラム３０はデコミットするポインタを戻り値に基づいて調整することができる。別の方法として、ファイル・システム２８が自身で残余を追跡し、アプリケーション・プログラム３０が部分的ファイル・セクションの残余をデコミットすることをあとで要求したとき、そのバイトをデコミットすることも可能である。

上記とは別に、好ましい方法は、アプリケーション・プログラム３０が事前定義のファイル・セクション・サイズを使用することである。従って、以下の例は好ましいＡＰＩコール条件に基づいている。さらに、６４キロバイトのような代表的なファイル・セクション・サイズを取り扱うとき、オフセット・パラメータに対応するクラスタはエクステント・リスト内に深く埋め込まれる可能性があるために、そのあとでデコミットするクラスタの数が相対的に大きな数になることがある。このような大きな数を扱うことを避けるために、以下の例ではその目的上、特に断りがない限り、任意のファイル・セクションのサイズがクラスタのサイズに等しいものとしている。つまり、どの特定クラスタもデコミット可能になっている。若干実用目的に合わないが、以上から理解されるように、このような小さなファイル・セクション・サイズの使用が可能であるため、以下の例は単純化されているが、そのことは本発明の精神と範囲から逸脱するものではない。

図３はファイル・ハンドルが５であるファイル４２$_1$の場合のエクステント・リスト５８を示している。第１の例では、アプリケーション・プログラム３０はＡＰＩ　４６をコールし、２０キロバイトのオフセットから始まり、８キロバイトの長さにわたって、ハンドル５をもつファイルをデコミットすることを要求している。この例では、クラスタ・ファクタ（cluster factor）はクラスタ当たり４キロバイトであり、１つのファイル・セクションは４キロバイトのサイズになっている。ＡＰＩ　４６がコールされると、図１２のステップ１００でＡＰＩ　４６はファイル・ハンドルを使用して、エクステント・リスト５８などの該当ファイル情報を選択する。ステップ１０２で、ＡＰＩ　４６は、２０キロバイト・オフセットをクラスタ当たり４キロバイトのクラスタ・ファクタで除することによって、そのオフセットをファイルの先頭から５クラスタ離れたオフセットに変換する。エクステント・リスト内の最初のエントリはクラスタ１０、長さ５であるので、ファイル・システムは、クラスタ１０、１１、１２、１３および１４が最初の５クラスタとして、この順序でファイル４２に割り振られていることを知っている。従って、オフセット値は第２のランのクラスタ１９を指している。ステップ１０４で、長さパラメータは８キロバイトをクラスタ当たり４キロバイトのクラスタ・ファクタで除することにより２クラスタに変換される。従って、図３に示すように、クラスタ１９と２０がデコミットされることになる。

次に、ステップ１０６で、デコミットするクラスタ、この例ではクラスタ１９と２０がラン境界上に置かれているかどうかが決定される。言い換えれば、ステップ１０６では、クラスタ１９がランの先頭にあり、クラスタ２０がランの末尾にあるかどうかが決定される。この例では、クラスタ１９と２０はランを開始し、終了しているので、このプロセスはステップ１１０にブランチする。

ステップ１１０で、クラスタ１９と２０はメタデータを修正することにより、具体的には、クラスタ１９と２０がもはやファイルの一部でないことを示すようにエクステント・リ

スト５８の値を変更することによってデコミットされる。図４に示すように、マイナス１（-1）の値は実クラスタ番号１９をオーバライトするが、有効なクラスタ番号を表していない、どの事前定義値でも表示値になるように選択することも可能である。最後に、ステップ１１２で、デコミットされたクラスタ１９と２０は空きスペースに戻される。ＮＴＦＳシステムでは、これはクラスタ１９と２０に対応するビットの値を、ビットマップ５２内でトグルすることによって行われる。ステップ１１０と１１２は、システム障害が発生したときオールオアナッシング（all-or-nothing）オペレーションを保証するように実行されることが好ましい。

図３および図５乃至図６は、デコミットされる１つまたは複数のクラスタがラン境界に正確に一致していない例を示している。この例では、エクステント・リストが図３に示すものと同じであり、図３と同じようにクラスタ・ファクタがクラスタ当たり４ＫＢ、ファイル・セクション・サイズが４ＫＢであるとして、クラスタ１３をデコミットしようとしている。上述したように、アプリケーション・プログラム３０は、５に等しいファイル・ハンドル（ファイル42_1）、１２キロバイトのオフセット、および４キロバイトの長さを指定してＡＰＩ　４６をコールすることによってかかるスペースの解放を開始する。同じく上述したように、これらのパラメータを使用して、ステップ１００－１０４はオフセットと長さパラメータをクラスタ情報に変換し、クラスタ１３だけがデコミットされるものと決定する。

しかし、ステップ１０６では、クラスタ１３がランに一致していないで、もっと大きなランの一部であると決定される。その結果、ステップ１０６はステップ１０８にブランチし、そこでエクステント・リスト５８内のメタデータが、図５に示すように、１０長さ５のランを１０長さ３、１３長さ１および１４長さ１の複数のランに分割することによって修正される。例えば、この修正はエクステント・リスト５８のランを一時スペースにコピーし、そのランをエクステント・リスト５８に戻すように再コピーし、新しい情報を挿入し、該当する場合には既存の情報を修正することによって行うことができる。なお、図５には事実上図３と同じクラスタがリストされているが、図５ではクラスタ１３はラン境界と一致している。

ステップ１０８に続いて、スペースは、クラスタ１３がファイル42_1にもはや割り振られていないことをエクステント・リスト５８に示すことによってステップ１１０でデコミットされる。クラスタ１３をデコミットしたことは図６の修正エクステント・リスト１３に示されているが、そこでは－１がクラスタ１３の個所に書かれている。最後に、ステップ１１２で、デコミットされたクラスタ１３はビットマップ５２を上述したように修正することによって空きスペース（ＮＴＦＳの）に戻される。

以上から理解されるように、メタデータの操作はエクステント・リストを通してではなく、マスタ・ファイル・テーブル５０内で直接的に行うことが可能である。図７は、図３、図５および図６を参照して上述したように、クラスタ１３がデコミットされたときそれに応じて、レコード54_1のデータ・フィールド56_1が直接に修正されると、ランがどのようになるかを示している。なお、図７には図６と同じ情報が示されているが、ここには、各ランの論理クラスタ値も示されている。

図８および図９は複数のランが１回のＡＰＩコールでデコミットされる例を示している。図８では、ゼロのオフセットから始まって、図３の最初の９クラスタは１回のＡＰＩコールまたは一連のＡＰＩコールで、ＡＰＩコールによってすでにデコミットされている。図９では、デコミットされたクラスタは連続するデコミット・クラスタの長さを合計することによって、単一の連続デコミット・スペース・ランに結合されている。本発明では必要でないが、このような最適化を定期的に行うと、エクステント・リスト５８内の（またはデータ・フィールド56_1内の）エントリ数が大きくなり過ぎるのを防止することができる。

本発明の一側面によれば、デコミットされたクラスタ・ランをエクステント・リスト５８から単に除去するだけではなく、ファイル・システム２６はそのランにデコミットされたとのマークを付ける。その結果、デコミットされたスペースはまだファイル44_1と論理

的に関係づけられているので、ファイル４２$_1$を基準にした、アプリケーション・プログラム３０のオフセットとポインタ値は有効のままになっている。例えば、アプリケーション・プログラム３０がファイル４２$_1$の先頭からの３２キロバイトのオフセットをファイル・システム２８に送るときはいつでも、オフセットは、デコミットされたファイル・セクションがあるかどうかに関係なく、正しい物理クラスタ（図３または図４のどちらも、クラスタ５０）を指している。

しかし、アプリケーション・プログラムの内部データ・ポインタが未変更であっても、アプリケーション・プログラム３０は戻って、デコミットされたスペースを読み書きすることができない。この目的のために、アプリケーション・プログラム３０がファイル・システム２８に読取り操作を要求したが、渡したオフセットまたは長さがデコミットされたスペースに対応するものであれば、ファイル・システム２８はそのスペースがすでにデコミットされていたと、ファイルのメタデータから判断し、デコミットされたファイル・セクションのバイトに対してゼロ（さもなければエラー）を戻してくる。事実、ＡＰＩ　４６が試みとして”WriteZeros（）”と名づけられているのは、アプリケーション・プログラム３０は特定ファイルのデコミットされたスペースを実効的にゼロにするからである。なお、デコミットされたスペースは空きスペースになるので、他のファイルで使用することができ、そこには非ゼロのデータが入るのが代表的であるが、デコミットされたスペースはそのスペースがデコミットされたファイルでは論理的にゼロだけを収めている。一部の高度セキュア・ファイル・システムでは、空きクラスタを物理的にゼロにしている。

同様に、あるファイルのデコミットされたスペースにデータを書き込もうとすると、そのファイルではエラーとみなされる。しかし、ファイル・システム２８に新しいスペース（これは偶然にデコミットされたスペースであることがある）を割り振らせ、そこにデータを書かせることも可能である。そのようにすると、ファイル・システム２８は新しく割り振られたスペースのロケーションとサイズを反映するように、そのファイルのメタデータを調整すること、つまり、新しいクラスタ・ランを正しいロケーションでメタデータに挿入すること（該当する場合には既存のランを延長すること）も必要になる。

代替ファイル・システムがアロケーション単位をファイルにマッピングするもう１つの方法は、ブロック・リストによるものである。公知のように、ブロック・リストは単純にファイルに関連するアロケーション単位のリストである。例えば、ＦＡＴシステムでは、ファイルの各クラスタはファイルの次のクラスタ（またはファイル末尾マーカ）を指しているエントリをファイル・アロケーション・テーブルの中にもっている。その情報から、種々データ・クラスタ間の関係をマッピングしているブロック・リストはそのファイルに関連するバッファに（または持続的メタデータ・ストレージに）置かれる。ファイル４２$_1$の場合の例示ブロック・リスト６０（これはＮＴＦＳシステムでは図３のエクステント・リスト５８に相当する）は図１０に示されている。

図１０は、ディスク・ボリューム上で利用可能な空きスペースを示すブロック・リスト６２も示している。このブロック・リスト６２内の情報はメモリに置いておくことも、ディスク・ドライブに置いておくこともできるが、どの場合も、ファイル・システム２８が利用できるようになっている。

図１１は、ファイル４２$_1$のブロック・リスト６０およびファイル・ハンドル５、オフセット２０ＫＢおよび長さ８ＫＢのパラメータを指定してWriteZeros（）ＡＰＩ　４６がコールされた後の空きスペースのブロック・リスト６２を示している。前述したように、図１２のステップ１００は該当するブロック・リストを選択し、ステップ１０２はオフセットをクラスタ１９を指すポインタ（先頭から５クラスタ）に変換し、ステップ１０４はデコミットする長さを２クラスタに変換する。しかし、ステップ１０６（場合によっては、ステップ１０８も）が実行される代わりに、ブロック・リストを使用するファイル・システムでは、ステップ１０４は即時にステップ１１０にジャンプする。これは、各クラスタが個々にリストされるとき（固有長さは１になっている）取り扱うべきランがブロック・リストになく、従って、デコミットが実効的に常に境界に一致しているからである。

ステップ１１０で、クラスタ１９と２０のブロック・リスト６０内のエントリはそこにイ

ンジケータ（－１など）を入れることによってデコミットされる。ステップ１１２で、クラスタ１９と２０は図１１のブロック・リスト６２に示すように、空きスペースに追加される。

使用されるのがブロック・リストであるか、エクステント・リストであるかに関係なく、アプリケーション・プログラム３０ではなくファイル・システム２８を使用してスペースのデコミットを管理する利点は、多くのファイル・システムが、ＮＴＦＳを含めて、システム障害から保護するセーフガードの働きをすることである。例えば、ＮＴＦＳはオールオアナッシングのトランザクションを保証するロギング・ファイル・システムである。つまり、各オペレーションはアトミック単位で実現されている。あるトランザクションの完了前にシステム障害が起こると、ＮＴＦＳはそのログを使用してオペレーションをロールバックする。

以下では、本発明をマージ・アプリケーション・プログラム（例えば、アプリケーション・プログラム３０はマージ・アプリケーションである）で利用した場合について説明するが、その概要図は、図１３乃至図１６と図１７のフロー図に示されている。公知のように、マージ・アプリケーション・プログラムは、２つまたはそれ以上のソート・ソース・ファイルからのデータを１つのソート・ターゲット・ファイルに結合するものである。例えば、アプリケーション・プログラムは、名前とアドレスのアルファベット順リストを同種の別のリストとマージして、ソース・リストの結合サイズと等しいアルファベット順リストを得ることができる。

マージ・アプリケーション・プログラム３０の予備ステップ２００（図１７）では、少なくとも２つのソース・ファイル７０、７２（図１３）がオープンされ、ターゲット・ファイル７４（図１３）が作成される。スペースを節減するために、以下で説明するように、ターゲット・ファイル７４は最初は１ファイル・セクションだけのサイズで、例えば、６４キロバイトでオープンされる。なお、前述したように、アプリケーション・プログラム３０は対応するファイル・システム２８のファイル・セクションのサイズを知っている。ステップ２０２はソース・データが読み取り可能になっているかどうかをテストして決定する。この時点ではソース・ファイル７０、７２はオープンしたばかりであるため（およびこの例では、長さが非ゼロであるため）、読み取るデータがあるので、ステップ２０２はステップ２０４にブランチする。ステップ２０４で、最初のデータ・ブロック（例えば、２０キロバイト）が各ソース・ファイル７０、７２から読み取られ、データを公知のようにマージすることによって処理される。この例では、マージ・データはサイズが３６キロバイトになる場合がある。なお、１つのソース・ファイルから処理されるデータ量は別のソース・ファイルから処理されるデータ量と同じでないのが一般である。従って、ループを通るたびに、追加データはステップ２０４で特定ソース・ファイルから必要になったときだけ読み取られる。

図１３は、２つのソース・ファイル７０、７２をターゲット・ファイル７４にマージする初期ステージにおけるマージ・アプリケーション・プログラム３０を示し、そこでは陰影を付けたエリアはファイル内のデータを表している。単純化のために、ソース・ファイル７０、７２は、それぞれが４つのファイル・セクションと５つのファイル・セクションからなるものとして示されているが、ソース・ファイルはどのサイズであっても構わず、必ずしもファイル・セクションが正確に倍数であるとは限らない。アプリケーション・プログラム３０は第１ソース・ファイル７０の先頭（図１３ではゼロ・バイト）を示す第１ポインタ７６または類似手段と、ソース・ファイル７０からデータが読み取られた（または下述するようにバッファに置かれ、処理され、および／または書き込まれた）個所を示す第２（読取り）ポインタ７８または類似手段とを保持している。同様に、アプリケーション・プログラム３０は第２ソース・ファイル７２の先頭（図１３ではゼロ・バイト）を示す第３ポインタ８０と、ソース・ファイル７２からデータが読み取られた個所を示す第４（読取り）ポインタ８２とを保持している。アプリケーション・プログラム３０はデータがターゲット・ファイルに書き込まれた個所を追跡するためのターゲット（書込み）ポインタも保持している。

ステップ２０６は処理されたデータ（４０キロバイト）がターゲット・ファイル７４の残余サイズに収まるかどうかをチェックして決定する。この例では、ターゲット・ファイルはオープンされたばかりであるので、６４キロバイト全部が使用されて処理データの３６キロバイトで満たされることになる。従って、プロセスはステップ２１０にブランチする。十分なスペースがターゲット・ファイル７４に残っていなければ、ステップ２０６はステップ２０８にブランチし、そこでターゲット・ファイル７４のサイズはファイル・システムに追加ディスク・スペースを要求することによって増加されることになる。なお、必要時にだけターゲット・ファイル７４にスペースを追加するようにすると、非常に大きなターゲット・ファイルを初めに作っておかなくてもマージを行うことができる。

どの場合も、ステップ２１０では、処理されたデータはターゲット・ファイル７４に書き込まれる。以上から理解されるように、システム障害から保護するために、バイトの読み取りを追跡するのではなく、ポインタ７８と８２はデータが各ソース７０、７２から読み取られ、処理され、無事にターゲット・ファイル７４に書かれた個所を追跡することができる。これにより、バッファに置かれたデータがシステム障害発生時に失われることが防止される。従って、ポインタ７８、８２はその時点でそれに応じて増加させ、図１３乃至図１６の右に移動することができる。例えば、図１４に示すように、ポインタ７８と８２は図１３のその位置に対して右に移動している。

ステップ２１２（図１７）は、１つのファイル・セクション・サイズ（この例では、６４キロバイト）より大きいデータがソース・ファイル７０から読み取られたか（または処理されて無事に書かれたか）どうかを決定する。例えば、データがどれだけソース・ファイル７０から読み取られたかを決定するために、プロセスはまずポインタ７８とポインタ７６との差を計算する。図１４に示すように、ポインタ７８と７６との差は１ファイル・セクションより大きくなっている。従って、ステップ２１４で、ファイル７０内で”１”で示されているファイル・セクションが解放される。

プロセスのステップ２１４は、上述したようにWriteZeros（）ＡＰＩ　４６をコールすることによってファイル７０の前方からファイル・セクションをデコミットする。この目的のために、ＡＰＩ　４６には、ファイル７０のファイル・ハンドル、オフセットのバイト数（図１４ではゼロ）およびデコミットする長さ（６４キロバイト、つまり、１ファイル・セクション）が渡される。同じく上述したように、WriteZeros（）ＡＰＩ　４６はファイル７０に対応するメタデータを操作することによってこれを行う。なお、データはマージ・アプリケーションでは順次に読み取られ、処理されるので、ファイル・セクションは前方からデコミットされる。

図１５は図のファイル・セクション”１”が読み取られた後のファイルの状態を示している。ポインタ７６はファイル・セクション”２”の先頭まで、例えば、６４キロバイトだけ進められ、ファイル・セクション”１”がデコミットされたことを示している。アプリケーション・プログラム３０は、「ゼロ」がいま解放されたファイル・セクション”１”の先頭をまだ示すようにそのポインタとオフセットを保持しているが、ファイル・システム２８はそのスペースに空きのマークを付けている。

ステップ２１４に続いて、ステップ２１６はソース・ファイルのすべてがテストされたかどうかを決定し、すべてがテストされていなければ、ステップ２１８にブランチして次に評価すべきソース・ファイルを選択する。従って、この例では、ソース・ファイル７２はまだテストされていないので、ステップ２１６はステップ２１８にブランチし、そのあとステップ２１２が再び実行されるが、この時はファイル７０のポインタ８０、８２が使用される。しかし、図１４に示すように、およびステップ２１２で決定されたとおりに、ファイル７２では、ポインタ８２とポインタ８０との差は１ファイル・セクション以下である。従って、スペースは解放されないで（今回はステップ２１４はスキップされる）、プロセスは直接にステップ２１６に戻る。テストする必要のあるソース・ファイルは残っていないので、プロセスはステップ２０２に戻り、追加のデータを読み取る。

図１６はさらに以後のステージでのマージを示している。図１６に示すように、ソース・ファイル７０と７２はターゲット・ファイル７４の増加と共に縮小している。その結果、

ファイル・システム２８は以前にソース・ファイル７０と７２に割り振られていたスペースをターゲット・ファイル７４に追加することができる。以上から理解されるように、この手法によると、ファイル・システム２８はどの時点でも、少量の一時スペースを割り振るだけで済むことになる。事実、ファイル・セクションがそのソース・ファイルから除去されたかどうかを、なんらかの方法でアプリケーション・プログラム３０がチェックして確かめるとした場合、最悪のケースでは、総一時スペース量はソース・ファイルの数にファイル・セクションのサイズをかけたものにほぼ等しくなる。最善のケースでは、総一時サイズは１ファイル・セクションに若干の増加分を加えたものである。

最終的には、ファイル７０と７２のどちらからも読み取るべきソース・データがなくなることになる。その時がくると、ステップ２０２（図１７）はステップ２２０にブランチし、そこで残っているソース・ファイルのスペースがあれば解放されることになる。例えば、ソース・ファイル７０と７２が削除される。

次に、図２４のフロー図に概要を示すように、持続的ＦＩＦＯキューで利用したときの本発明について説明する。図１８は以前にデキューされたが、ファイル８６から消去されていないデータ量（セクション”Ａ”で示されている）をもつ持続的ＦＩＦＯキュー・ファイル８６を示している。ここではそのように示されていないが、”Ａ”はゼロ・バイトであってもよい。ポインタ８８または類似手段はデキューされたスペースが始まるロケーション（例えば、バイト・オフセット）を追跡し、別のデータ・ポインタ９０または類似手段はファイル８６のセクション”Ｃ”内のデキューされていない残余データ・アイテムの先頭を追跡する。なお、スペースが上述したように以前にデコミットされていれば、デキューされたスペースの先頭はアプリケーション・プログラム３０側から見たときバイト・ゼロになく、このことはデータ・ポインタ８８内の非ゼロ値によって示される。例えば、ポインタ８８は４５０キロバイトの位置を示すことがあるのに対し、ポインタ９０は４７５キロバイトを示すことがある。

DeQueueオペレーションが図２４のステップ３００で実行されると（これにより１つまたは２つ以上のアイテムがファイル８６の前方から除去される）、別のデータ量がデキューされている。この追加のデキュー・データは図１９にセクション”Ｂ”として示されている。以上から理解されるように、デキュー・オペレーションの一部として、ポインタ９０は残余の非デキュー・セクション”Ｃ”データの新しい開始位置を示すように調整される。この例では、ポインタ９０は５００キロバイトの位置を示すように増加されている場合がある。以上から理解されるように、図２４に示すプロセスは各アイテムがデキューされた後ではなく、いくつかのデキュー・オペレーションの後で実行することができる。

デキュー・オペレーションに続いて、ステップ３０２で、デキューされた総スペース量が、例えば、ポインタ８８にストアされたバイト値をポインタ９０にストアされたバイト値から差し引くことによって計算される。この総サイズは”Ａ”と”Ｂ”の結合セクションで図１９に示されており、この例では、５０キロバイト（つまり、５００キロバイト－４５０キロバイト）に等しくなっている。ステップ３０４で、この総デキュー量は図１８乃至図２０にインターバル”ＦＳ”で示されているファイル・セクションのサイズと比較される。この例では、ファイル・セクションはサイズが６４キロバイトであり、これは固定され、アプリケーション・プログラム３０に知らされている量である。５０キロバイトはファイル・セクションの６４キロバイトより小であるので、この時点ではスペースはデコミットされず、ステップ３０４はブランチして図２４のプロセスから出る。しかし、次回に図２４のプロセスを通るときは、セクション”Ａ”からスタートするのではなく、ポインタ９０はセクション”Ａ”と”Ｂ”の両方がデキューされていた個所より先に進んでいる（つまり、この例では５００キロバイトのオフセット）。その結果、ＦＩＦＯキュー・ファイル８６は以前にデキューされたが、ファイル８６から消去されていない新しいデータ量（つまり、５０キロバイト）になっており、これは図２０にセクション”Ｄ”で示されている。

図１８乃至図２０とは対照的に、図２１乃至図２３は総デキュー・スペースがファイル・セクションより大である場合を示している。以前と同じように、図２１は以前にデキュー

されたが、ファイル８６から消去されていないデータ量（ゼロ・バイトのこともある）をもつＦＩＦＯキュー・ファイルを示している。このデータはセクション”Ｅ”で示され、例えば、７００キロバイトから始まり、７３０キロバイトマイナス１バイトにわたっている場合がある。なお、図２１乃至図２３では、ポインタ８８と９０は図１８乃至図２０を参照して前述したものと類似のデータ・ポインタ情報を保持しており、この例では、それぞれ７００キロバイトと７３０キロバイトの値をもっている（従って、その値を指している）。

図２２に示すように、Dequeueオペレーションが図２４のステップ３００で実行されたあと、セクション”Ｅ”で示すように別のデータ量がデキューされている。この場合も、ポインタ９０は残余の非デキュー・セクション”Ｇ”の新しい開始位置を示すように調整され、例えば、７８０キロバイトに増加されている。しかし、図２２に示すように、今度は総デキュー・スペース（”Ｅ”＋”Ｆ”）は１つのファイル・セクション、つまり、この場合もインターバル”ＦＳ”で示されている図２１乃至図２３のファイル・セクション・サイズより大になっている。

従って、ステップ３０２で総デキュー・スペース量は、例えば、ポインタ８８（７００キロバイト）をポインタ９０（７８０キロバイト）から差し引くことによって計算され、８０キロバイトが得られる。ステップ３０４でこの総デキュー・スペース量（８０キロバイト）はアプリケーション・プログラムによってファイル・セクションの既知サイズである、６４キロバイトと比較される。しかし、今回はデキュー・スペースはファイル・セクション・サイズより大であるので、ステップ３０４での比較はステップ３０６にブランチする。

本発明によれば、アプリケーション・プログラムはファイル８６の前方からスペースをデコミットするファイル・セクションでＡＰＩをコールする。例示のコールはWriteZeros（5, 700KB, 64KB）にすることができ、ここで”５”はファイル・ハンドルである。なお、持続的ＦＩＦＯキューのアプリケーション・プログラム３０は、デキュー・スペースをファイル・セクションのサイズで除するだけで２つ以上のファイル・セクションをデコミットすることができ、その商はデコミットするファイル・セクションの数と等しくなる。別の方法として、プロセスは残余のデキュー・スペースが１つのファイル・セクション以下になるまでループバックすることによって、ファイル・セクションを１つずつ繰り返しデコミットすることも可能である。

ステップ３０８で、ファイル・システム２８に関連するＡＰＩ　４６は該当数のファイル・セクションを解放し、解放したスペースを利用可能ディスク・スペースに加える。これは、前述したようにファイルのメタデータと空きスペース・ビットマップなどを操作することによって行われる。ステップ３１０で、アプリケーション・プログラム３０はポインタ８８を、デコミットされたなかった残余デキュー・スペースの先頭に移す。この例では、ポインタ８８は７６４キロバイト（７００キロバイト＋６４キロバイト）に増加する。この増加を分かりやすく示したのが図２３であり、セクション”Ｈ”はデキューされたが、ファイルから消去（デコミット）されていない残余スペースを示している。

以上から理解されるように、ファイル８６の末尾以外から、この例では前方からスペースを解放すると、高速にかつ安価な方法で持続的ＦＩＦＯキューを消去することができる。事実、この消去操作は、データをコピーする必要もなければ、複雑なファイル操作の必要もなく、必要ならば、アイテムがデキューされるつど実行できるくらい効率的なものである。

しかし、消去操作によると物理ファイル・サイズは保たれるが、論理ファイル・サイズは持続的ＦＩＦＯキューや場合によっては他のアプリケーションによると大きくなっていく。例えば、ファイルの前方はスペースがデコミットされると縮小するとしても、論理ファイル・サイズは新しいアイテムがキューに追加されると大きくなっていく。また、ポインタはその値が大きくなっていく。これは、ファイル・サイズとポインタが論理的に2^{64}バイトまで大きくなることができるＮＴＦＳでは問題にならないことが認められているが、他のファイル・システムでは、最終的には、この成長を続ける数を取り扱うとき問題が起

こる可能性がある。
その結果、本発明ではその必要はないが、このような暴走的成長の可能性を認識し、望ましいときにファイル・サイズとポインタを実効的にリセットするようにアプリケーション・プログラムを書くことができる。この目的のために、アプリケーション・プログラム３０は有用（デキューされていない）データを一時ファイルにコピーし、オリジナル・ファイルが占有しているスペースが解放される。つまり、オリジナル・ファイルは削除される。そのあと、メタデータは一時ファイルを指すように再マッピングされ、ポインタ８８、９０はゼロにリセットされる。実際には、ファイルの前方にあるホールが除去される。
ある種のデータ・コピーを行うと、上記のような最適化が得られるが、この最適化は頻繁に行う必要はない。例えば、このような最適化を行う望ましい時期は残余の有用データが相対的に少ないときであり、大量にコピーする必要がないからである。もう１つの望ましい時期は、ファイル・サイズまたはポインタの値がある種の、あらかじめ決めたしきい値の安全量を超えたときである。
その内部ポインタを調整するアプリケーション・プログラム３０に関連してファイル・システム２８にこの最適化を行わせることも可能である。例えば、アプリケーション・プログラム３０による要求が別のＡＰＩを通して行われたとき、デコミットされたクラスタのランをファイルのメタデータから除去し、ファイル・サイズをそれに応じて調整することができる。同様に、ＦＡＴシステムなどのファイル・システムでは、最初のクラスタを指している情報および／またはＦＡＴテーブル・エントリはデコミットされたクラスタをスキップするように修正することができる。デコミットされたスペースがファイルの元の先頭から連続しているときは、メタデータとアプリケーション・プログラムの両方のゼロ・ポイントがシフトされる。ただし、ここで注目すべきことは、このような最適化は、デコミットされたデータがファイルの先頭で連続していないときは、アプリケーション・プログラム３０が各不連続の、デコミットされたファイル・セクションごとにそのポインタを調整する必要があるため複雑化することである。
詳しく上述した説明から理解されるように、ファイル・システムにおいてファイルの任意の論理部分に割り振られたディスク・スペースを解放する方法とメカニズムが提供されている。この方法とメカニズムはディスク・スペースの解放を高速化すると共に、大量のデータをコピーしたり、大量の一時スペースを割り振ったりしないで済むようにする。この方法とメカニズムはファイル・システムに組み込まれ、ファイル・システムが具備する既存セーフガードと一緒に機能してシステム障害から保護し、マージ・アプリケーションや持続的ＦＩＦＯキューを取り扱うアプリケーションなどの、アプリケーション・プログラムで容易に利用することができる。この方法とメカニズムは高速、単純で、信頼性があり、拡張可能であり、ほとんどどのファイル・システムでも有効に働く。
本発明は種々態様に変更し、代替構成も可能であるが、上記ではいくつかの実施例を図面に示して、詳しく説明してきた。しかし、本発明は上述した特定実施形態に限定されるものではなく、その中には本発明の精神と範囲に属する、すべての変更や改良、代替構成、および等価技術が含まれるものである。

【図１】

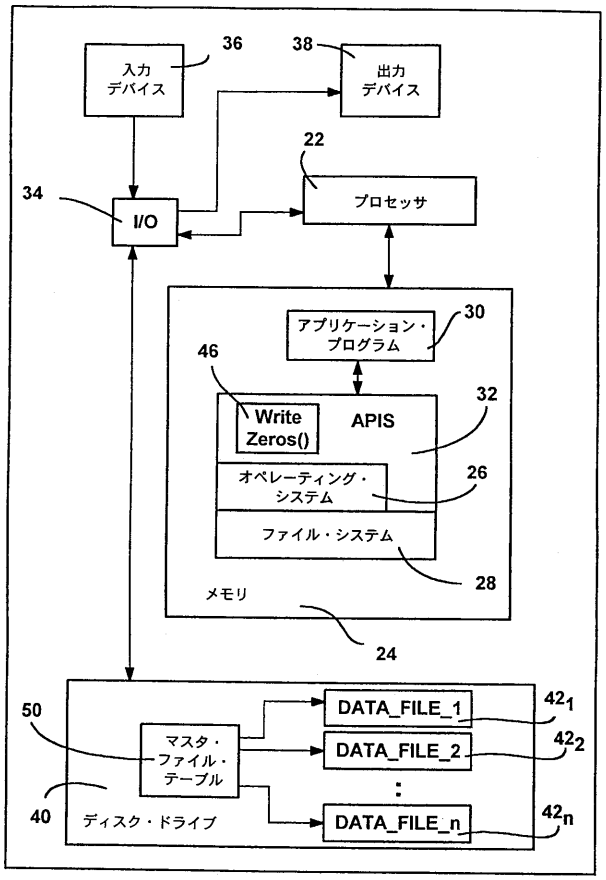


FIG. 1

【図２】

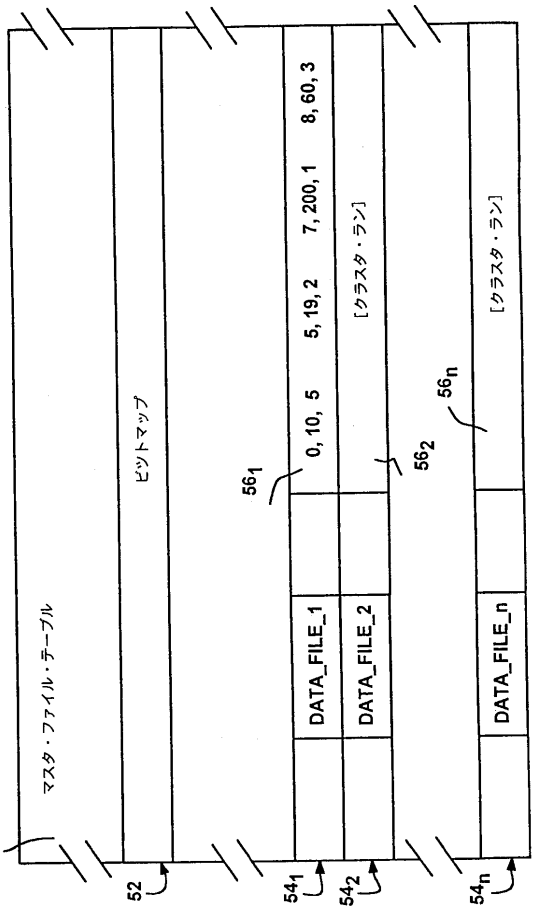


FIG. 2

【図３】

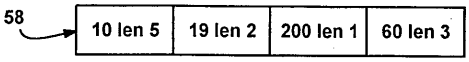


FIG. 3

【図４】

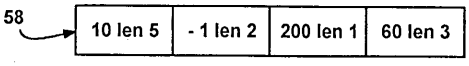


FIG. 4

【図５】

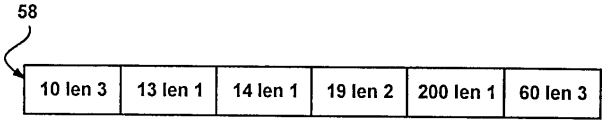


FIG. 5

【図６】

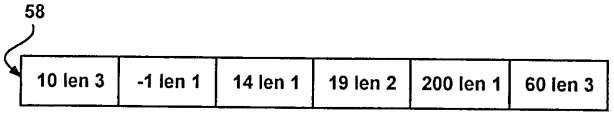


FIG. 6

【図８】

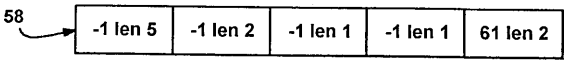


FIG. 8

【図９】

FIG. 9

【図７】

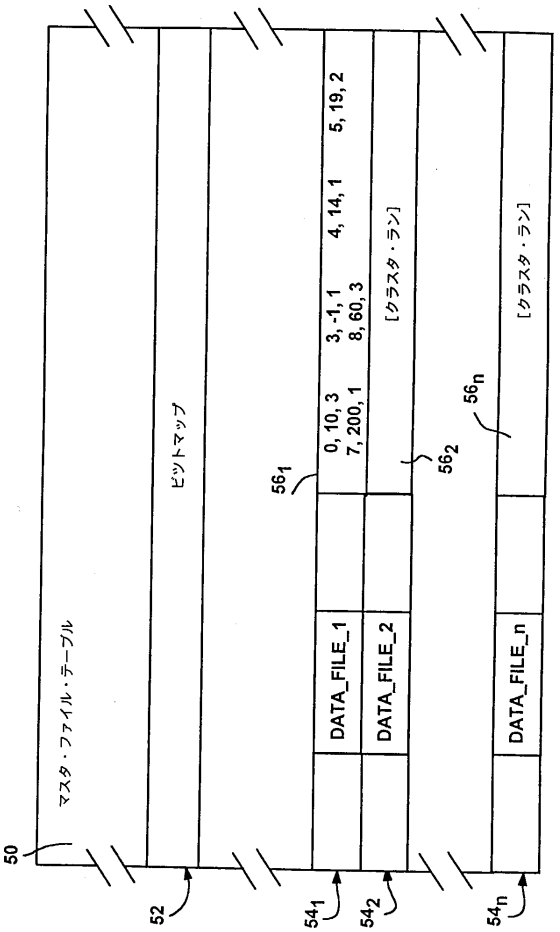


FIG. 7

【図１０】

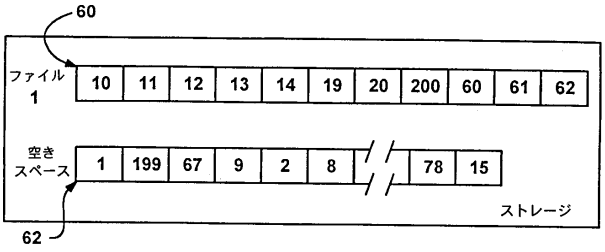


FIG. 10

【図１１】

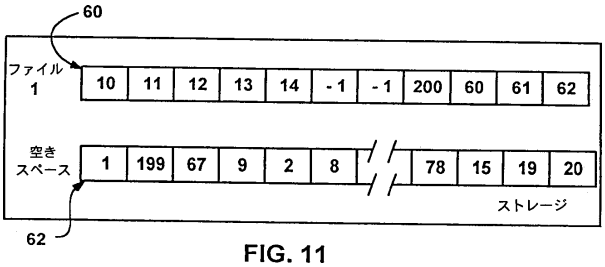


FIG. 11

【図１２】

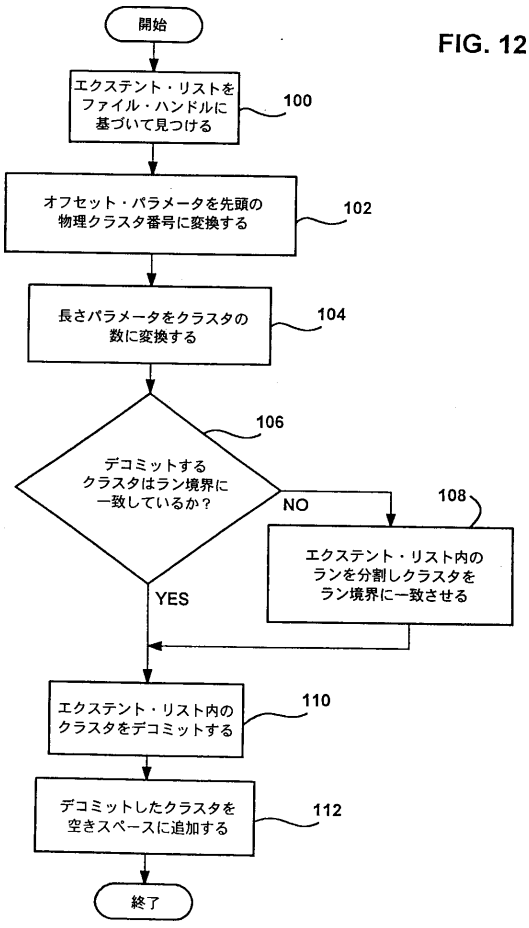


FIG. 12

【図１３】

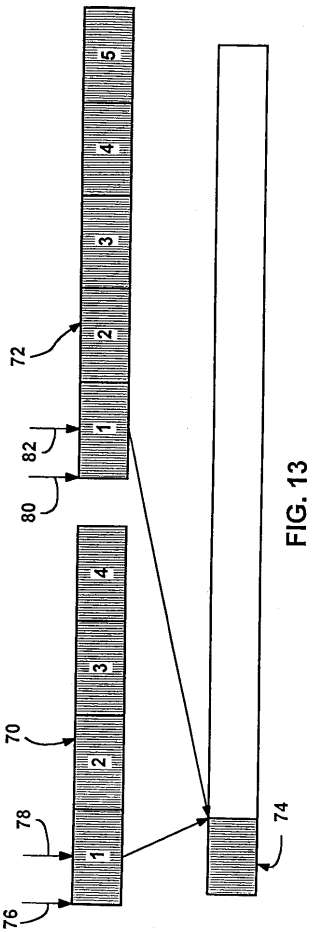


FIG. 13

【図１４】

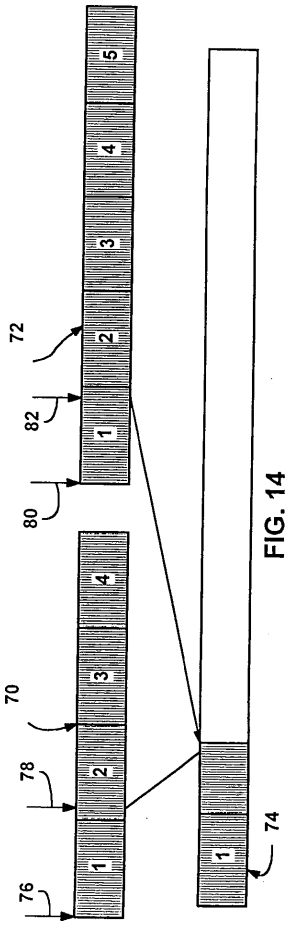


FIG. 14

【図１５】

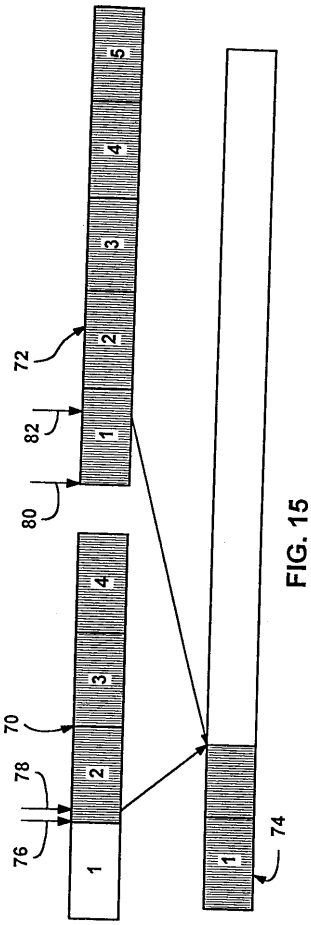


FIG. 15

【図１６】

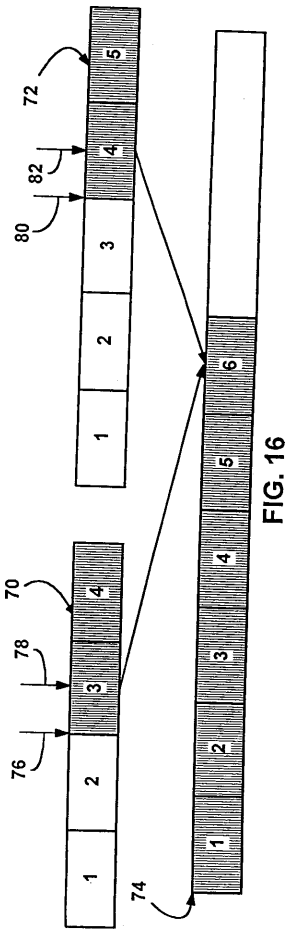
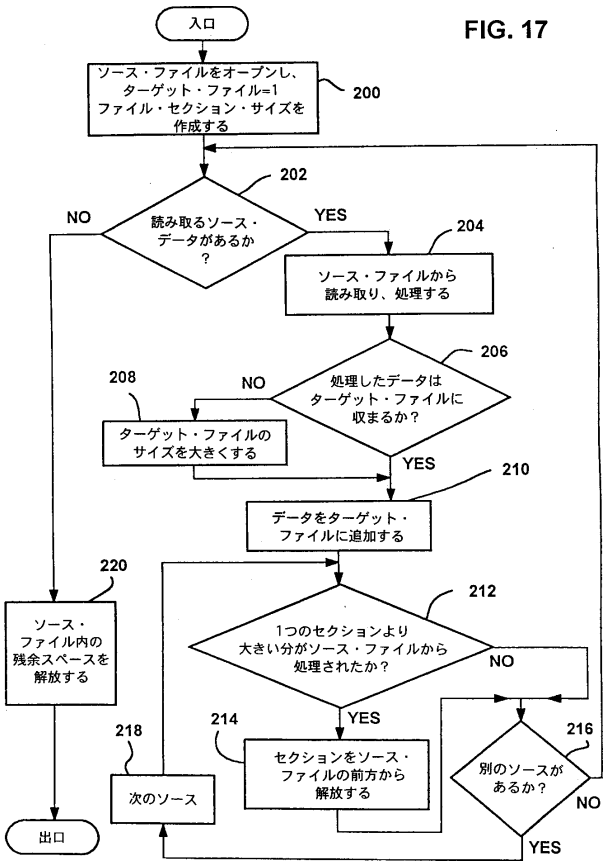


FIG. 16

【図１７】

FIG. 17



【図１８】

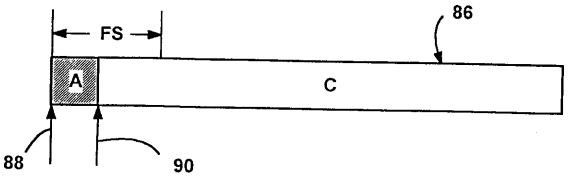


FIG. 18

【図１９】

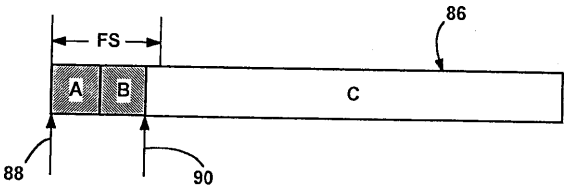


FIG. 19

【図２０】

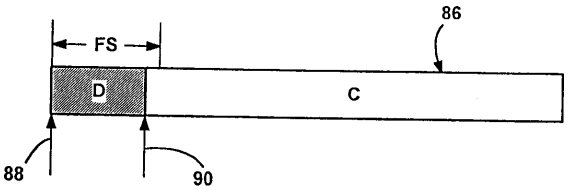


FIG. 20

【図２１】

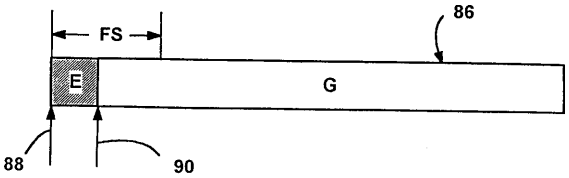


FIG. 21

【図２２】

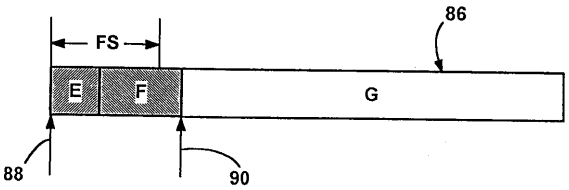


FIG. 22

【図２３】

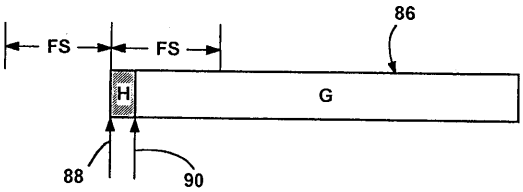
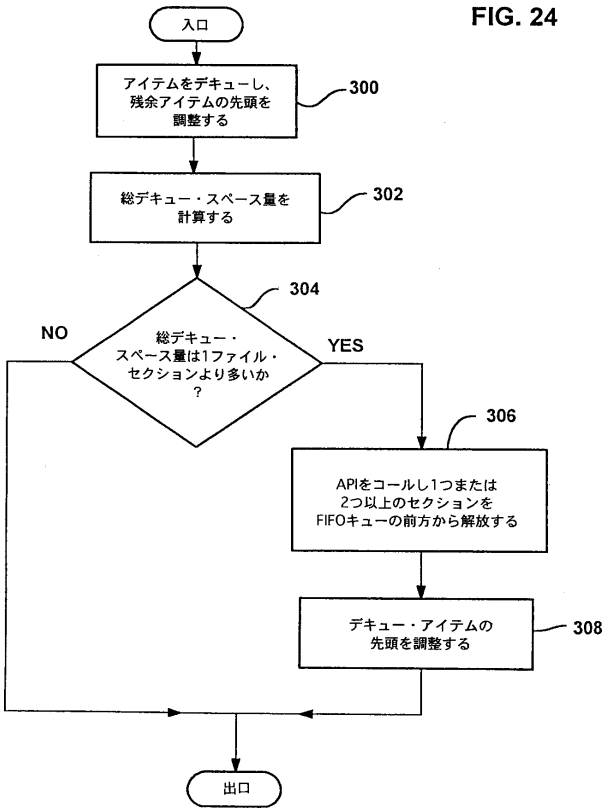


FIG. 23

【図２４】

FIG. 24



フロントページの続き

(72)発明者 ペルトネン，カイル，ジー.
アメリカ合衆国 ９８０２７ ワシントン州 アイザッカー サウスイースト ４２エヌディー プレイス １８１２６
(72)発明者 バーコウィッツ，ブライアン，ティー.
アメリカ合衆国 ９８００７ ワシントン州 ベルヴィー １４２エヌディー プレイス ノースイースト ３９１２
(72)発明者 ズビコウスキー，マーク，ジェイ.
アメリカ合衆国 ９８０７２ ワシントン州 ウッディンヴィル ノースイースト １７８ティーエイチ プレイス １５８１７
(72)発明者 マイルスキー，バートズ，ビー.
アメリカ合衆国 ９８１２５ ワシントン州 シアトル４６ ティーエイチ アヴェニュ ノースイースト １００５２

審査官 原 秀人

(56)参考文献 特開平０８－３２８９７０（ＪＰ，Ａ）
国際公開第９６／１８９６０（WO，Ａ１）
Maurice J. Bach，ＵＮＩＸカーネルの設計，日本，共立出版株式会社，１９９１年 6月１０日，第1版，ｐ．９５－１０２
Ａ．Ｖ．エイホ 外2名，情報処理シリーズ１１ データ構造とアルゴリズム，日本，培風社，１９８７年 3月10日，第1版，ｐ．４８－５３
ＭＶＳ／ＥＳＡ アプリケーション開発解説書：アセンブラー呼出し可能サービス（ＯｐｅｎＥｄｉｔｉｏｎ MVS用）ＭＶＳ／ＥＳＡシステム・プロダクト，日本アイ・ビー・エム株式会社，１９９６年 5月，第5版，ｐ．170-172，735-738，1059-1060
Samuel J.Leffer 外3名，The Design and Implementation of the 4.3BSD UNIX Operating System，米国，Addison-Wesley Publishing Company,Inc.，１９８９年，ｐ．２９－３４

(58)調査した分野(Int.Cl.，ＤＢ名)
G06F 12/00